AFRL-IF-RS-TR-2005-171
Final Technical Report
May 2005

RESEARCH IN ARCHITECTURAL APPROACHES TO THE INTEGRATION OF EMPIRICAL, ANALYTIC, AND EPISODIC LEARNING WITHIN SOAR

The Regents of the University of Michigan

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. Q109

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-171 has been reviewed and is approved for publication

APPROVED: /s/

RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: /s/

JOSEPH CAMERA, Chief
Information & Intelligence Exploitation Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAY 2005	Final Sep 03 – Sep 04

4. TITLE AND SUBTITLE
RESEARCH IN ARCHITECTURAL APPROACHES TO THE INTEGRATION OF EMPIRICAL, ANALYTIC, AND EPISODIC LEARNING WITHIN SOAR

5. FUNDING NUMBERS
C - F30602-03-2-0261
PE - 62301E
PR - COGV
TA - 00
WU - 01

6. AUTHOR(S)
John E. Laird

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Regents of the University of Michigan
DRDA
3000 South State Street
Ann Arbor Michigan 48109

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFED
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2005-171

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Raymond A. Liuzzi/IFED/(315) 330-3577/ Raymond.Liuzzi@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The goal of this research as originally stated:
"We propose to greatly expand the use and usefulness of the Soar architecture. Thus, our goal is to create a highly useable and available cognitive architecture that will greatly increase the use of cognitive architectures. To date, cognitive architectures have been either research or proprietary software. Soar has been one of the most successful cognitive architecture for developing knowledge-rich performance systems. We will greatly improve the development environment; develop a new, integrated debugger; expand and improve the tutorial and technical documentation; and provide training and support of the Soar architecture. All software will be freely available on SourceForge."

During this project we have made significant progress in improving the Soar architecture and building a community.

The major thrust of our work can be divided into three parts:
1. Evaluate the current state of Soar software, supporting documentation and tools.
2. Create a plan for future improvements to Soar and development of any new tools.
3. Make significant progress on the improvements to Soar and its associated tools and documentation.

14. SUBJECT TERMS
Artificial Intelligence, Cognitive Architecture, Machine Learning

15. NUMBER OF PAGES
23

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1. Evaluation of current state of Soar software

Our evaluation led to the establishment of eight subprojects described below.

1.1 SOAR KERNEL, MAINTENANCE AND SUPPORT.

The Soar Kernel is the core Soar software. It is written in C and has been extensively modified through the years as new capabilities have been added. Overall the kernel is still solid; we only minor changes and clean up required at the current time. There are some extensions that should be made, but we did not have the resources to pursue them under this proposal.

1.2 SOAR KERNEL, FORMAL SPECIFICATION.

Although there is extensive documentation on the Soar Kernel, there does not exist a formal specification of the system. During the development of the current Soar code base (in the early 1990's) a formal specification of Soar was created in Z. However, the specification was very low level and was not executable, making it impossible to check if the specification was consistent with the implementation. We determined that creating a new formal specification was important.

1.3 SOAR KERNEL INTERFACE.

Although the underlying Kernel is in relatively good shape, we determined that another significant improvement could be made in the way Soar interfaces to other environments and tools. Currently Soar is written in C and has an interface system called SGIO. SGIO works well with C or C++, but is not a complete solution for interfacing the kernel to other environments and tools. Soar Technology had started development of a new approach called gSKI. This would make it easier to quickly interface with new tasks. We determined that it would be extremely beneficial to the Soar community to adopt the gSKI approach. Unfortunately, gSKI was originally a proprietary extension to Soar. After intensive discussions, Soar Technology agreed to "donate" gSKI to the Soar community and move it to a BSD license.

1.4 SOAR DEVELOPMENT AND DEBUGGING ENVIRONMENT + BENCHMARK TASKS

We have a set of independent editing (Visual Soar) and debugging (TSDebugger) tools. Visual Soar is written in Java while the TSDebugger is written in TCL/Tk. In addition we have a set of task environments that are used for teaching Soar (and for overall research): Eaters and TankSoar. Both of these are written in TCL/Tk. We determined that in the long run we need to move to a more unified environment, and selected ECLIPSE an open source Java-based integrated development environment (IDE). This transition will require creating new tools and environments.

Below is a list of specific tools and addition changes we determined were needed:

1.4.1 Visual Soar (the editor for Soar):

Visual Soar is the editor used to develop Soar. We collected and organized all bug and enhancement reports in Bugzilla and prioritized all bugs. We have polled users as to what the most important enhancements are and they are being implemented.

1.4.2 Soar Debugger

The original debugging environment is called the Tcl/Soar Interface (TSI). We have identified shortcomings in it and are in the process of completely revamping it (to create the TSDebugger).
In the long-term, however, we have decided that we will write a new debugger from scratch in Java.

1.5 SOAR WEBSITE & FACILITATE INTERACTION WITH THE SOAR COMMUNITY

We need to create an easier to maintain web site and one that is easier to navigate.

1.6 SOAR MANUAL

We completely reviewed the current manual and created prioritized bug and enhancement lists.

1.7 SOAR TUTORIAL

We determined that the Soar tutorial needs to be update and made more accessible to novices.

2. To do lists for future improvements to Soar

Each component of Soar now is assigned to a specific person for maintenance and development (these components are all listed in the next section where we describe the progress we've made in them). We also track all bugs for Soar, VisualSoar (the Soar development environment), and SGIO (the Soar General I/O system) on SourceForge.

3. Improvements in Soar, Associated Tools, and Documentation

Below is our list of subprojects, associated support personnel, and progress during the last three months.

3.1 SOAR KERNEL, MAINTENANCE AND SUPPORT: JONATHON VOIGT AND ROBERT MARINIER

We have made some improvements to the kernel. The most important is the ability to have decisions based on probabilities. We knew that there were a few serious bugs in the kernel, but they had never been identified because of their seemingly transitory nature – they only arose in our larger systems that involved interaction with dynamic environments and thus they were difficult to reproduce. We were able to identify and fix these bugs so that there are currently no outstanding critical bugs in Soar. Moreover, all bugs are now tracked using Bugzilla. This provides web access to our bug lists and a uniform process for tracking all Soar bugs..

In mid-November 2003, we released Soar 8.5, which included significant new capabilities for Soar, fixed all known crashes in Soar, and fixed critical bugs in Visual Soar (the Soar editor). This release also incorporated a new installer that greatly simplifies the installation of Soar on Windows and Mac platforms.

In June 2004 we made a new release of Soar (8.5.1). This included a significant redesign of the current Soar Debugger (written in Tcl/TK). We also were able to fix the last of the priority 1 bugs with Soar – there are currently no known crashes. This release also included an improvement to the tutorial, the manual, and the demos. In July 2005 we released Soar 8.5.2 which corrected a few bugs and is now the current release.

3.2 SOAR KERNEL, FORMAL SPECIFICATION: ROBERT MARINIER

We have just started on creating a formal specification of Soar in ASML (Abstract State Machine Language), an executable specification language that is supported by Microsoft. Microsoft is using this for some of its projects and ASML is very well supported (there are plug-ins for it even in Word and .Net). The specification will be much simpler and easier to understand than the C implementation because it abstracts away from many implementation details. One important aspect of the ASML specification is that it is executable. Once we have a working specification, we can compare its execution to the actual code so that we can ensure that the specification is consistent with the current implementation. (The specification is executable, but not optimized, so it will probably run orders of magnitude slower than the C version.) The specification work has also highlighted a few implementation details which may be at odds with the theory (or at least deserve further discussion). In the future, we can experiment with changes to the architecture using the formal specification before writing actual C code. At this point, the formal specification will become the gold-standard for the working Soar architecture. This will make it possible for users and developers to have an unambiguous representation of what the code should be.

3.3 SOAR KERNEL INTERFACE.

Over two years ago, Soar Technology started development of a separate branch of Soar (based on Soar 8.3.3). In doing so, they concentrated on the interface between Soar and other systems as well as a reorganization of the Soar code. We entered into discussions with Soar Technology and they agreed to “donate” their enhancements into the mainline, open source release. Over the last three months, we have been concentrating on merging our research version of Soar (8.5.2) with Soar Technology's version of Soar (8.3.3+). We have worked extensively with one of Soar Technology's kernel programmers, Jens Wessling, during this process. The kernel merge has involved creating reports detailing the differences between the two kernels and porting bug fixes from the research kernel into the Soar Tech kernel along with many new features including but not limited to numeric indifference preferences, capture replay, match time interrupts, and the removal of unchecked buffers that can cause buffer overflow situations.

We have finished the merger portion of the work, but we discovered in doing this work that much more work needed to be done than we originally expected, in part because of our lack of understanding as to what Soar Tech. had completed. However, the end result will be a very

flexible structure that makes it possible to connect to Soar in a variety of ways. These include in process, across process, and across machine, all determined at run time. The intermediate communication interface is based on XML, and we are providing support through SWIG so that it should be possible to connect Soar to many other languages. Initially we are targeting JAVA and C++, but the SWIG interface should support a large variety of other languages such as TCL and Python. The structure will also make it possible to dynamically link and unlink development tools to Soar so that we can jump in on processes that need to be monitored without paying the costs of monitoring all processes. A significant aspect of our design has been to maintain the efficiency of Soar, so for example, for most interactions between Soar and an external domain, XML is not actually created, only data structures are passed, and there is very little inherent overhead to the communication.

3.4 SOAR DEVELOPMENT AND DEBUGGING ENVIRONMENT + BENCHMARK TASKS

3.4.1 Visual Soar (the editor for Soar): Andrew Nuxoll.

We have prioritized all bug reports for Visual Soar and started to work through the bugs and enhancements (over 20 fixed). We also evaluated the use of Jemacs as a replacement for the current home-grown editor in Visual Soar. Unfortunately this proved to be a failure as Jemacs was not mature enough for our needs. In contrast, we evaluated the use of Eclipse as a possible framework for reimplementation of Visual Soar. The pilot re-implementation was successful and we are pursuing a more complete reimplementation in parallel to supporting the current version (just to be safe).

3.4.2 Soar Debugger: Douglas Pearson (Three Penny Software) and Mazin Assanie

In addition to fixing several high-priority bugs, we developed a new TS-debugger that offers a much cleaner, streamlined interface with multiple user-configurable views into the agent's internals. Moreover, it provides several conventions commonly found in debugging utilities for iterative paradigms, for example breakpoints and memory watches.

In the long-term, we will write a new debugger from scratch in Java. We have created an initial requirements document for the new debugger and have a preliminary mockup of the design and interface.

3.4.3 Benchmark Tasks (Eaters and TankSoar):

Over the summer of 2004 we reimplemented our benchmark tasks (Eaters and TankSoar) in Java.

3.5 SOAR WEBSITE AND SOAR COMMUNITY : JONATHON VOIGT

We converted all of our web pages to Sitemaker – a tool that makes it much easier to modify and maintain websites. We reorganized pages to make it easier to find documentation and programs. The new Soar web site is at "http://sitemaker.umich.edu/soar/". Our activities include moving all maintenance of the Soar SourceForge.Net collaboration environment including repository backups and administration, maintenance of the new Soar bug tracking system located at

"http://winter.eecs.umich.edu/soar-bugzilla/", mailing list migration to SourceForge.Net, mailing list moderation, administration of a local server used for Soar games research, and design and deployment of the Soar Workshop 24 web site and custom registration system located at "http://winter.eecs.umich.edu/workshop24/".

We established the Soar Consortium to provide a more structure process for proposing, implementing, and maintaining the Soar kernel. More details are available on request. We also hosted the 24th Soar Workshop in June in Ann Arbor.

3.6 SOAR MANUAL: SHELLEY NASON

We completely reviewed the current manual and created prioritized bug and enhancement lists. We added text for new 8.5 features and have started to rewrite and extend specific chapters.

3.7 SOAR TUTORIAL: JOHN LAIRD

We created a completely new introductory exercise for teaching Soar. This includes over 40 pages of text, software for hands-on running, and slides to teach the tutorial. The tutorial was given at the Soar Workshop, the International Conference on Cognitive Modeling, and the National Conference on Artificial Intelligence. Each tutorial had between 14-20 participants. Feedback was extremely positive and we have been "invited" back to AAAI in 2005 (and will be repeating the tutorial at the Soar Workshop this summer – there is no ICCM this year).

3.8 DEMONSTRATION SOFTWARE:

We have converted all demos of Soar over to 8.5 and the newest version of Visual Soar.

4. Future of Soar Development

Overall, this project has allowed us to significantly enhance the Soar architecture, making it easier to learn, use, and maintain. However, we still have significant work to complete our migration to ECLIPSE as an integrated development environment and using the new gSKI approach for interfacing Soar. Both projects will completed by July 2005. Our work on gSKI has been made available in a beta release to researchers and is playing a significant role in our integration with TIELT (a DARPA funded project on creating an integrated machine learning and reasoning testbed/environment.

Appendix I Draft of First Chapter of a Book on Soar

This appendix gives an overview of the philosophy behind Soar. It is a draft of the first chapter of a book on Soar that is under development.

Artificial Intelligence. Few topics inspire as much imagination, wonder, controversy and possibly anxiety as Artificial Intelligence (AI). Human-level AI systems are the computers or robots that you dreamed about when you first heard of AI: HAL from "2001, a Space Odyssey"; Data from "Star Trek"; or CP30 and R2D2 from "Star Wars". From the first days of computer science, writers have explored the possibilities of human-level AI, starting with Turing's seminal paper on "Computer Machinery and Intelligence" in 1950, and continuing on with Newell, Simon, Minsky, McCarthy, Hofstadter, Kurzweil, and Moravic. In science fiction, the writings of Isaac Asimov, Robert Heinlein, and Philip Dick explored what it meant to be human by exploring what computers could be.

Although completely autonomous robots or androids may be the ultimate in human-level AI, we don't expect to see them for many years. Nonetheless, it may be possible to create AIs, while not human-level in all aspects, that have a much broader range of capabilities than today's AIs. Already, autonomous AI systems participate in large-scale military training exercises, standing in for what would otherwise be very expensive humans. Possibly the most visible use of AI (besides being the evil opponent in movies) is in computer and video games, where computer controlled opponents and teammates interact with human players, enriching virtual worlds.

Given the long-standing popular and scientific interest in human-level AI, one would think that it is at the center of AI research, with researchers at universities and industrial labs toiling late into the night trying to build and program computers that mirror or exceed human mental capabilities across a broad range of tasks. This is hardly the case. Over the last thirty years, the majority of researchers and developers in AI have abandoned dreams of general human-level AI and focused on developing special-purpose algorithms for specific problems. This has led to important applications of AI and significant progress in many of the subfields of AI, but there is little evidence of progress on the problems of building agents with all of the mental capabilities of humans.

HUMAN-LEVEL AI

What is it that distinguishes humans from what is currently achievable in artificial intelligence? First, humans appear to effortlessly draw on all of their cognitive capabilities for a given task. They appear to seamlessly integrate whatever capabilities they need, easily moving from using their personal experience, their knowledge about the world, to using a variety of reasoning and planning strategies. They are able to process huge amounts of incoming data, picking out what is relevant to the task at hand, communicating with other humans when necessary. That is not to say people are perfect, but there is a significant contrast between human behavior and AI systems where the norm is to have at most one or two of these capabilities, and where the integration is usually anything but seamless. Thus, one challenge for developing human-level AI is to create ways of building systems where these capabilities can synergistically co-exist – for example where language could be used during planning to get external help on a knotty problem or a

situation where more information is required, and conversely, where planning could be used to help generate a novel and complex utterance.

A second and more fundamental distinguishing characteristic of humans is their ability to work on an incredibly diverse set of problems using whatever knowledge they have available. Humans are not always brilliant nor can they solve every problem they encounter, but they have the ability to attempt and usually make progress on a wide variety of tasks. Moreover, they improve both from practice and from drawing on whatever knowledge sources are available. In contrast, most AI systems are designed for solving one type of problem using a pre-specified knowledge. Even systems that do learn are invariably limited to one problem or a predefined class of problems. AI as a field has been very successful outside of achieving general human-level AI. We do have human-level or even super-human-level AI systems for specific tasks, such as for playing checkers or chess, searching through credit card transactions to discover fraud, helping you search through the internet, solving problems in computational biology, or scheduling aircraft departures at airports. But we don't have AI systems that can work on a wide variety of problems, seamlessly integrating the capabilities we associate with human intelligence.

This book is about one research effort directed toward understanding what is required for human-level AI and developing systems that embody that understanding. The effort is centered on a *cognitive architecture* call Soar. An architecture comprises the fixed structures, the building blocks, for creating intelligent behavior. Soar reflects, as do many other cognitive architectures, the assumption that a single uniform approach to reasoning underlies all cognition. This book presents the structure of Soar, the rationale for its design, examples of how it has been used to develop agents, an analysis of it as an architecture for developing human-level AI, and comparisons with other architectures. Central to our approach is that we continually build systems using Soar so that we can test out new ideas in the architecture and learn whether Soar is sufficient for a new set of tasks or problems. Although Soar falls short of supporting human-level intelligence in many ways (see chapter N for more details), it is remarkable in its ability to integrate knowledge-intensive reasoning, reactive execution, hierarchical reasoning, planning and learning. The tasks it has been applied to span a range from standard expert systems, to natural language processing, to synthetic pilots for military training simulations, to characters in computer games. On the one hand, Soar represents a long-term research project, where we continually attempt to extend the human-level capabilities we are able to achieve in a computer, but on the other hand, Soar is a specific architecture that you can use for developing AI systems today. If you are interested in the theory of Soar, read on. If you are interested in building human-level AI systems, also read on. In the end, you may not decide to use Soar for your application, but we hope you will learn from the ideas in Soar and they will help you no matter what approach you take.

SOAR HISTORY

Our own pursuit of human-level AI builds on the ground-breaking research of Allen Newell and Herb Simon, who in 1956 created the first running AI program, the Logic Theorist. In the sixties, Newell and Simon, together with their colleagues, developed computer programs that could solve problems by heuristically searching through the problem space of possible states. The General Problem Solver (GPS) was remarkable because it broke with the tradition of developing a computer program to solve a single problem. Using means-ends analysis, it could solve a dozen

different problems if it was given a formal description of the problem and how the steps in the problem related to the goals. In the late sixties and early seventies, Newell and Simon also were the first to explore using rules to encode the knowledge for problem solving, developing the first rule-based system, PSG. Research on efficient matching techniques led to the first widely used production system architecture, OPS-5, which is the ancestor of many production rule languages including CLIPS, ART, ECLIPSE, and JESS, as well as Soar.

In the early 1980's, John Laird, under the guidance of Allen Newell, attempted to take a step beyond GPS by creating a system that could use not just a single method, such as means-ends analysis, but whatever method was appropriate to the knowledge it had about a task. The approach taken was to merge the ideas of heuristic search in problem spaces with rule-based systems as the representation for knowledge. The system was originally built as an extension to XAPS-2, an experimental production system being developed to explore learning by Paul Rosenbloom, also a student of Allen Newell at the time. The basic cycle of the system consisted of selecting states, applying operators, and generating new states, all controlled by the rule-based knowledge. This led to naming of Soar, which originally stood for state, operator and result.[1] In Soar, as in GPS, there is a strict distinction between the fixed, primitive structures, such as the rule matcher and a decision procedure for selecting operators, and the knowledge, which can vary from task to task. Thus, Soar is the fixed structure, and we refer to it as the underlying *architecture* in the systems that are built with it. We will return to a general discussion of cognitive architecture following this brief history of Soar.

Using this first version of Soar, we encoded a variety of simple tasks and demonstrated that as knowledge was added, the methods it used would change. The methods were all simple methods (such as hill climbing, means-ends analysis, progressive deepening) that do not have complex internal structures (such as the simplex method), and thus we claimed that this original version of Soar realized a *universal weak method* (Laird & Newell, 1983). To improve its efficiency, Soar was rewritten as a modification of the OPS5 language, and an automatic subgoaling mechanism was added. This was followed by the addition of chunking, an adaptation of ideas from Paul Rosenbloom's thesis work. At this point, we (John Laird, Allen Newell, and Paul Rosenbloom) decided to attempt a more real-world task. We chose to re-implement an existing expert system, called R1 (later renamed XCON). R1, developed by John McDermott, configured VAX computers for Digital Equipment Corporation. At the time, it was one of the largest expert systems in existence. Our project, spearheaded by Paul Rosenbloom, successfully recoded a substantial fraction of R1 into Soar. Instead of using a shallow encoding of the task with many special-case rules, the Soar implementation had a much more general understanding of what was involved in computer configuration. Initially, this *deep* problem solving approach was computational expensive, but chunking compiled the deep approach into rules that allowed R1-Soar to approach the efficiency of the original R1 system. Throughout the remainder of the 1980's, there were many explorations into how chunking in Soar could lead to different learning styles, including strategy acquisition, learning macro-operators, and semantic learning. For example, David Steier built a serious of systems that attempted to discover new algorithm and then transfer knowledge learned in one problem to another.

[1] Although "Soar" started as an acronym, we now use it as a proper name.

Two other significant paths emerged in the late 1980's. The first was the reconceptualization of Soar not just as an architecture for building AI systems, but also as a model for human behavior. Soar models were developed for a wide range of applications including natural language parsing, playing simple video games, solving puzzles, performing the tasks of the NASA test director, visual search and identification tasks, and simple concept acquisition. As follow-on to his previous work in model human problem solving, Newell proposed that cognitive psychology should pursue unified theories of cognition (UTCs), that is, broad computationally based models of the primitive mechanisms that he hypothesized underlie human cognition (Newell, 1990). Thus, his hypothesis was that the architecture underlying cognition is uniform and can be used as the basis for a theory of cognition. In his book, *Unified Theories of Cognition*, Newell used Soar as an example of what such a theory would look like (although recognizing that it fell short in many ways).

The second, parallel branch involved using Soar for tasks that required interaction with external environments. All of the original tasks that Soar was used for were completely *internal* tasks, where all of the problem solving involved the manipulation of internal representations. To explore external interaction, researchers at the University of Michigan interfaced Soar to a Puma robot and camera system and then a Hero robot. They also created simulations of these environments, and interfaced Soar to the SGI flight simulator. These explorations led to significant changes in the Soar architecture, directly related to how Soar maintain consistency between its internal representation of state and its perception of its external environments. Following that redesign, there was a complete re-implementation of Soar in C (Soar was originally written in Lisp). This not only made Soar much more efficient (the C version ran 20 time faster than the Lisp version, in large part because of new algorithms and data structures, not just because of the change in language) and more portable (Soar could now run on any platform and did require the purchase of Lisp).

Fresh from the improvements in efficiency, we once again revisited the issue of working on a real-world task, this time much larger and challenging than R1-Soar. Beginning in 1992, we began developing synthetic pilots for use in large scale military simulation training exercises. These systems emulate the behavior of military personnel performing missions in fixed-wing (TacAir-Soar) and rotary-wing (RWA-Soar) aircraft and have since been used in large-scale military training exercises (Jones et al., 1999; Tambe et al., 1995). TacAir-Soar and RWA-Soar agents are autonomous, making decisions based on their awareness of the surrounding environment, commands received from other entities, and their extensive knowledge of military doctrine and tactics. They have the ability to pursue mission objects alone, or they can participate in larger groups made up of other synthetic agents or even human teammates participating via simulators (Laird, Jones, & Nielsen, 1998).

Since the late 1990's, Soar systems have continued to be developed across a wide range of domain that include both cognitive modeling and AI applications. A company, Soar Technology, Inc., was spun off of the efforts of developing TacAir-Soar at the University of Michigan and continues to develop Soar systems for modeling and simulation applications. For example, in a joint project between University of Michigan and Soar Technology, we developed adversary agents for Military Operations in Urban Terrain (MOUT) as shown in

Figure 1. In these cases, the environment for the Soar agents was created in a commercial computer game called Unreal Tournament. Humans could control avatars in the game, and engage multiple enemies, each controlled by Soar. The Soar agents could communicate among themselves and coordinate their behavior.

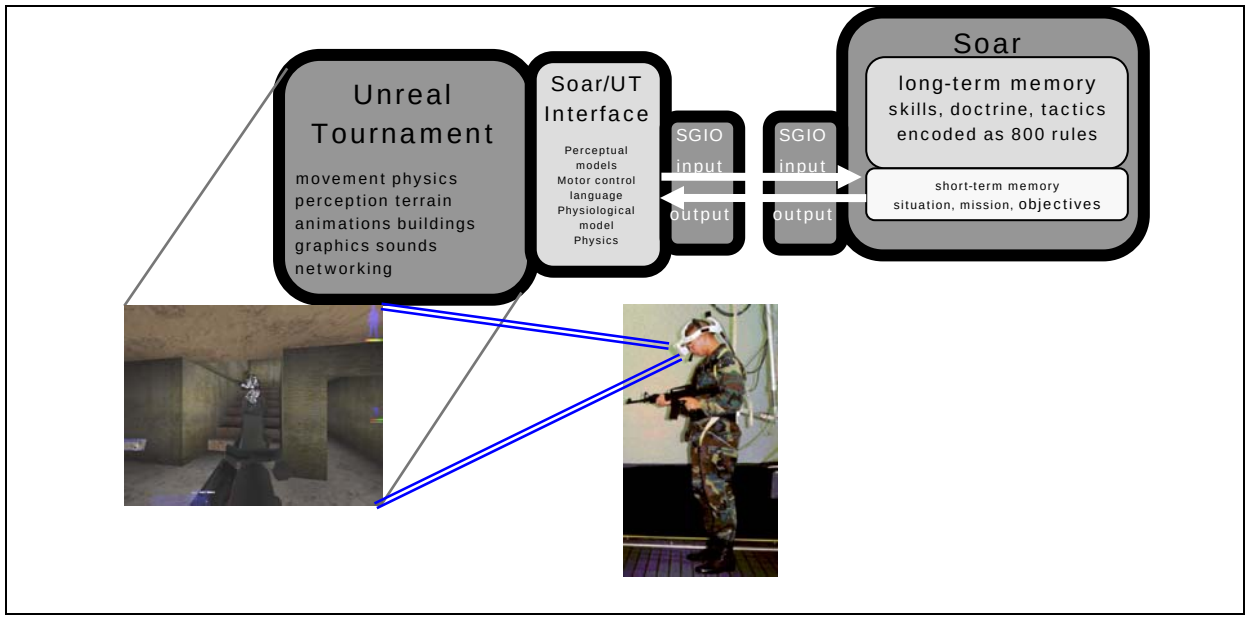

Figure 1: Soar Agents for MOUT

Work on natural language processing continues at Brigham Young University, while research on HCI applications of Soar is actively pursued at Penn State. The Institute for Creative Technologies (ICT) associated with the University of Southern California uses Soar as the integrating architecture for developing synthetic characters to use in non-combat military training systems. Also, the University of Michigan, Soar Technology, and ICT are actively involved in using Soar to control characters in computer games.

Starting in 2002, research started on some major extensions to learning in Soar. These reflected a realization that additional architectural learning mechanisms were needed in Soar. We have added reinforcement learning and episodic learning to Soar. We are also actively exploring the addition of a separate semantic learning module to complement these other learning mechanisms. All of these learning mechanisms are discussed in Chapter 5, along with an analysis of the advantages and disadvantages of adding independent architectural learning mechanisms in a cognitive architecture.

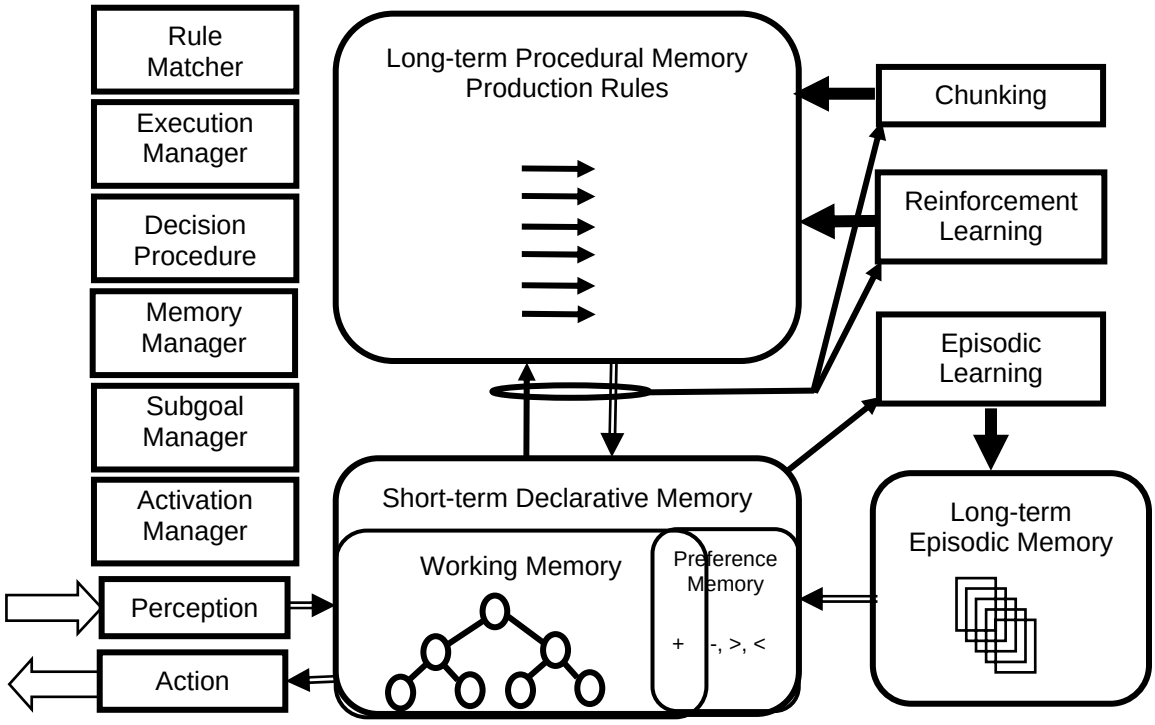

Figure 2: A block diagram of the Soar architecture

Figure 2 provides a high-level overview of Soar. It shows its basic structure with the memories represented as rounded-edged rectangles and the processes as square-edged rectangles and arrows indicating the transfer of data between components. Short-term memory holds representations of the current situation: current goals, percepts of the environment, assessment of the current situation in relation to its goals, intermediate results of reasoning, and so on. Behavior is driven by the matching of production rules (if then statements) in long-term procedural memory against working memory. The rules that match make changes to short-term declarative memory. Episodic memory holds snapshots of past short-term memories that can be deliberately recalled, which can then influence behavior by causing rules to match. On the left of the figure are the fixed processes that manage Soar operation, matching rules, making decisions, creating subgoals, and so on, while on the right are the learning mechanisms. Developing an agent in Soar involves writing rules that are stored in the procedural memory. Additional rules are learned by chunking and reinforcement learning.

Our goal for Soar is that supports the development of agents that can approach humans in their flexibility and adaptability in dealing with complex environments and different types of knowledge. At its core, Soar provides a framework for acquiring, organizing, and using its available knowledge to make decisions. We have attempted to be as broad in considering different sources of knowledge, be it programming by a human or learning from experience, where experience is broadly construed to include not just direct interactions with an environment, but also interactions with other agents (learning by instruction, learning by observation), and learning based on reflecting over its own knowledge.

Although Soar is an integrated architecture, with all of these components always available, there is spectrum of simple to complex agents that can be implemented in Soar based on which of the processing components are used. Simple agents can be created in Soar using the perception module, the action module, the rule matcher, the execution manager, the memory manager, and the decision procedure (as well as the short-term memory and the long-term rule memory), and Chapter 3 describes in detail how these components interact with each other and the memory components. Agents that use goals and meta-reasoning require the addition of the subgoal manager (as well as some extensions to the execution manager, the decision procedure, and the memory manager), which are described in Chapter 4. Soar has multiple learning mechanisms that use the activation manager (not a learning module itself, but activation is used to bias reinforcement and episodic learning in Soar), chunking, reinforcement learning, episodic learning, and episodic memory, which are described in Chapter 5.

Since 2003, we have also been exploring adding emotion and physiology to Soar (not shown in Figure 1). Specifically, we are creating a computational implementation of appraisal theory (## Lazarus and Smith). The impact of emotion on cognition is controversial and our research attempts to explore the impact that emotion can have on determining goals, learning, decision making, as well as being an interface between physiology and cognition. The embodiment of Soar agents, including physiology and emotion are discussed in Chapter 6.

More detailed examples of agents built with Soar are presented in Chapter 7. Chapter 7 also provides the venue for discussing issues that arise in building agents that have large collections of represented knowledge, teams of agents, and agents that learn. Soar provides guidance at one level as to how these knowledge representations should be organized, but this is true mostly in organizing representations that are used moment to moment. In agents with large knowledge bases, and that exist over long periods of time, there can be complex relationships between goals and tasks that have little to do with the underlying architecture. Researchers in the Soar community have developed a number of conventions for organizing knowledge representations for the knowledge-intensive agents that are discussed in this chapter.

COGNITIVE ARCHITECTURE

Fundamental to our approach to achieving human-level AI is the importance of the underlying architecture that supports cognition. In standard computers, the architecture consists of memories, processing and control components, and interfaces between these components and input/output devices. The memories hold data and programs, while the processing components execute the program instructions, performing the primitive computational processes, such as mathematical operations on numbers, moving data, or comparing values and changing which instructions are to be executed. The architecture also defines the language for specifying and controlling computation. In standard computers, the goals for that language are usually in conflict. On the one hand the language should be general and flexible so it is easy to specify any arbitrary computation, and on the other, it can be efficiently and reliably implemented, in terms of computation, cost, energy consumption, and so on. Often an architecture will support many different ways to do the same or similar things with different tradeoffs so that the programmer can choose the one that is best for the specific application. In once sense, the architecture design is itself an interface between the underlying hardware and the software that controls it.

Cognitive architectures are close cousins to computer architectures. They have memories, processing and control components, and interfaces, but on top of attempting to support general computation efficiently, they have additional constraints in how computation is performed. For one thing, instead of being *programmed*, we say, possibly presumptively, that a cognitive architecture uses its *knowledge* to produce behavior. This distinction is between the architecture and the agent's representations of knowledge, is also a distinction between the task-independent components of cognition – the primitive building blocks that are always available, and the task-dependent knowledge that adapts the agent to specific problems. A central idea of cognitive architecture is that it is the architecture itself (not the agent knowledge) that should support any properties and capabilities of human-level intelligence that are ubiquitous across all tasks. Such properties include: behavior that is inherently interruptible, many aspects of knowledge and behavior that are open to internal reasoning (meta-reasoning), and learning is possible almost every aspect of the behavior. These constraints are difficult to achieve by arbitrarily programming a standard sequential program architecture. Programs are difficult to change at run time and one instruction is executed after another, requiring that a programmer develop some higher level models of execution for the system. A cognitive architecture is exactly this, an attempt to specify a model of computation that directly supports intelligent behavior. Although cognitive architectures could be implemented directly in hardware, they are invariably software constructs, built through levels of interpretation on top of existing computers using languages such as C, C++, Java, or Lisp.

Below is a list of some of the architectures that have been developed in AI and cognitive psychology.

- ACTE through ACT-R (##Anderson, 1976; Anderson, 1993)
- Soar
- Theo (##Mitchell, 1985)
- Prodigy (##Minton & Carbonell, 1986; Veloso et al., 1995)
- PRS (##Georgeff & Lansky, 1987)
- CIRCA (##Munsliner & Atkins, 1993)
- 3T (##Gat, 1991; Bonasso et al., 1997)
- EPIC (##Kieras & Meyer, 1997)
- APEX (##Freed et al., 1998)
- 4D/RCS (##Albus)
- ICARUS (##Langley & Shapiro, 2003)
- Clarion (##Sun)

In terms of approaches to creating human-level intelligence, cognitive architectures are a middle ground between pure bottom-up and top-down approaches. A pure bottom-up approach would be to allow much less constrained organizations of knowledge, where the languages for specifying knowledge are much richer and more heterogeneous. At the extreme, this approach could use a general-purpose programming language such as C++, Java, or Lisp. A slightly more constrained approach is to include some specific support for more abstract knowledge representations. Similarly, one could also define a language of simple primitives that have little or no constraint on how they are combined. The research of Rodney Brooks seems roughly to match this description. In comparison with these bottom-up approaches, cognitive architectures constrain the form of knowledge and how knowledge can interact.

The advantage of using a cognitive architecture is that the structured forms of knowledge representation restrict the legal forms of knowledge that can be expressed in the system, which provide the following benefits:

1. Architecture can provide some universal performance properties that are hard to achieve with arbitrary programming
    a. Interruptability
    b. Learning becomes tractable because arbitrary programs do not need to be learned.
    c. Meta-reasoning is possible across all behavior.
    d. Hierarchical organization of procedural and declarative knowledge
    e. By restricting the form of knowledge, the architecture can prevent or at least detect non-productive behavior, such as illegal memory references (this should be true of any modern programming language, so I'm not sure about this).
    f. Provide default behavior when knowledge is missing.
    g. If done right, the primitive unit of knowledge is always productive – it isn't ½ of what is necessary to produce behavior. Tied to e-f.
    h. Possibly some performance guarantees for certain core functionality.
2. Common approach to all problems - eliminates ability to modify core functionality and have different approaches for different subproblems that don't work together when later make the decision to combine. Some argument about reuse – navigation and natural language code across agents. Can also create higher level development and debugging tools.
3. The promise is that it allows programmer to think at the level of task and knowledge and not details of implementation – make this more precise.

The disadvantages of using a cognitive architecture are:

1. Whatever is in the architecture cannot be changed – if something is not right for your application, you are stuck with it, or if something is missing, you either have to go without or try to add it yourself, or do it through *interpretation* – meaning it will have to be done with knowledge in the architecture. In a standard programming language, you could probably do it directly; however, the advantage of doing it through interpretation is that it is open to all of the features supported by the architecture such as meta-reasoning and learning.
2. Efficiency – often direct programming in a language such as C would result in faster, less memory intensive code.

The challenge for cognitive architecture designers in contrast to straight programming is to

1. Come up with the right set of primitives that are useful over the broad range of tasks humans engage in.
    a. Reasoning
    b. Learning – traditional learning systems are kept.
    c. Autonomy – doesn't crash, always can do something.
2. Create the primitives and mechanisms that guarantee certain types of behavior
    a. Interruptablity
    b. Meta-reasoning
    c. Learning

    d. Hierarchical organization of knowledge (procedural and declarative).
3. Implement them efficiently.

The top-down approaches focus on knowledge representations and not how they are used. In these approaches, the goal is for knowledge to be represented in ways that it can be used for many different applications, which invariably leads to declarative representations of knowledge. The Cyc project is the foremost example of this approach, although a significant core of AI has been devoted to formal knowledge representation. The goal of Cyc is to encode vast bodies of knowledge that could be used for many different tasks. These goals are somewhat different than that of cognitive architectures. Cyc is meant to be a component of an overall reasoning system, and thus it does not incorporate mechanisms for sensing, acting or decision making. Instead, a cognitive architecture could use it as a knowledge base, deliberately querying it for answers when knowledge is not directly encoded in architecture memories. In contrast, cognitive architectures are usually able to deploy their own knowledge much more efficiently because it is specialized for its use. In Soar, rules have conditions that determine when the associated knowledge (the actions) should be retrieved. In contrast, knowledge-level approaches usually require more extensive and less directed searches of their knowledge to determine what is relevant to the current situation.

RESEARCH METHODOLOGY

Given the overall goal of achieving human-level behavior, we have drawn heavily from both cognitive psychology and artificial intelligence in developing Soar. We take humans as the existence proof that general intelligent systems are possible and cognitive psychology informs us of the myriad of cognitive capabilities that humans possess that we know are sufficient for producing human behavior. More subtly, we also look to psychology to inform us of capabilities are not needed because humans have been successful without them, such as perfect memory or purely logical reasoning. From artificial intelligence, we draw on computational algorithms and data structures for acquiring, encoding, and using knowledge. Given that our goal is to *build* systems, we need the precision that comes with computational systems to specify behavior to a level where all the details are spelled out.

An alternative approach to developing human-level AI, that has many proponents in AI, is to ignore much of psychology and assume that there are many possible ways to achieve general intelligence. One possibility is that by taking an "engineering approach", where we design intelligence systems from scratch, we can do better than what evolution has achieved through trial and error. AI has had many successes in specific problems where AI systems meet or exceed human behavior such as planning systems, chess, and in depth data analysis using methods whose details diverge from human approaches. Nonetheless, for general intelligence, we assume humans are the place to start. This assumption makes sense not only because humans are very good, general intelligences, but also because developing human-level intelligence is going to be an incremental process. It seems unlikely (although not impossible) that general artificial intelligence will arise from one or two great breakthroughs. There will not be the key equation written on the blackboard, followed by the building of a robot, which is then is turned on and suddenly is completely humanlike. Instead, achieving human-level intelligence will be a long path of small incremental discoveries and ideas. We have chosen an incremental design strategy where we pick specific classes of problem, specific capabilities, and specific types of knowledge

as our starting points. We build systems that solve those problems using those capabilities and types of knowledge. We then expand to new problems, new capabilities, and new types of knowledge. If we ignore humans, we risk developing systems that work on one class of problems, but then completely fail when we try to extend them to the next level of capability. We might have to start over as the new constraints that must be incorporated into our design conflict with design decisions we made earlier. In contrast, if we are inspired by humans, we know that if we successfully capture a subset of the mechanisms and approaches they use, we assume those mechanisms and approaches are compatible with achieving broader, general intelligence – although it is still possible that our implementations have incorporated additional assumption that could require redesign. This path does not guarantee success, but it mitigates some of the risks. A secondary advantage to basing our design on an abstraction of human behavior is that our systems may be better able to interact with humans, both because humans are more likely to understand the operation of our agents and because our agents are more likely to understand humans**.**

Soar has been not only inspired by cognitive psychology, it also has been used to create detailed models of human behavior. Allen Newell (1990) hypothesized that architectures like Soar should form the basis of *Unified Theories of Cognition*. Such research continues in Soar as in related architectures, including ACT-R, Clarion and EPIC. Nonetheless, in this book our emphasis is on human-level behavior – achieving the capabilities and functionality of humans without detailed modeling of human data, such as reaction times and error rates.

We attempt add capabilities to Soar that have proven extremely valuable to humans. Many of these are well recognized by AI such as reasoning, meta-reasoning, planning, etc. Nonetheless, there are other human capabilities that AI has ignored that might prove extremely valuable in creating autonomous intelligent agents. For example, anyone who has seen the movie *Memento* gets an appreciation for how crippled humans are without an episodic memory, but this has all but ignored in AI, except for some work in case-based reasoning. In addition, evidence over the last two decades has pointed more and more to the importance of emotion to decision making and learning, but it also is largely ignored in AI.

Although we draw on cognitive psychology, we have made the tactical decision not to draw directly on neurophysiology. That is, we are attempting to build systems that work the same way as the human mind, but not necessarily the same way as the human brain at the level of neural circuits and neurons. In taking this approach we could fall victim to the problem described above – it is possible that by not modeling the human brain, we will come up with approaches for one set of problems and capabilities that can not be extended to include others. Moreover, there could be general properties of the human brain that we are completely missing as we incrementally attempt to cover more and more capabilities. So why don't we attempt to model the human brain? The first reason is that the behavior we are interested occurs at time scales of tenths of seconds and above, much above the range of neurons and neural circuits. Soar, and especially ACT-R, have demonstrated that it is possible to model human behavior without resorting to neurophysiology. A more practical reason is that we want to build agents right now, using current computers, which is possible with the types of models we develop in Soar, but would be infeasible if we attempted to model the neurons or even neural circuits. However, the most important reason is the mind, and computers, and cognitive architectures, are all symbol systems,

capable of universal computation. A lot of AI is predicated on the notion that symbol systems are the right abstractions for approximating systems that behave as if they have knowledge. Thus, it is not necessary (in a computational sense) to worry about the details of the neural level, such as how to achieve symbol composition with neural computation. However, while much of AI assumes this hypothesis (the Physical Symbol Systems Hypothesis, ## Newell 1980) it is a hypothesis and could be mistaken. But years of research suggest that a symbol system can approximate the human mind, and thus is a useful abstraction from which to attempt to build human-level systems incrementally.

FINAL THOUGHTS

Because the Soar project is necessarily incremental, evaluating and measuring progress can be challenging. It is relatively straightforward to evaluate individual applications or research systems, but clearly distinguishing what the architecture contributed from what the developer contributed is difficult. This inability to define the contribution of the architecture in a solution clearly makes understanding if new architectural mechanisms are needed (or if some of the existing ones superfluous) an empirical, rather than formal process. Soar researchers use Soar to build systems not only because Soar is a useful tool, but because these systems provide evidence of the strengths of the architecture and its weaknesses. While such evidence is admittedly weak, Soar has been used for hundreds of research systems and applications but its basic computational model has not changed during its existence. This constancy across such a range of applications suggests Soar offers some good ideas about the nature of general intelligence. In the remainder of the book, we will describe those properties of environments and tasks that have influenced the evolution of Soar, describe Soar as a cognitive architecture in detail, and describe how Soar is used to realize increasingly complex agent systems. We will also compare Soar to a number of other architectures, lay out some directions for future Soar evolution, and finally return to the subject of architectural research methodology and evaluation.

Appendix II: Soar Tutorial Introduction

This appendix is the introduction to the Soar Tutorial. The complete tutorial is available from: **http://sitemaker.umich.edu/soar/documentation_and_links.**

This is a guide for learning to create software agents in Soar, version 8. It assumes no prior knowledge of Soar or computer programming.

The goals of this document are:

- Introduce you to the basic operating principles of Soar.
- Teach you how to run Soar programs and understand what they do.
- Teach you how to write your own Soar programs.

This is about the nuts and bolts of writing Soar programs, but not about the theory behind Soar. For that, you should read Chapter 3 of the Soar Manual.

This tutorial takes the form of a sequence of lessons. Each lesson introduces concepts one by one and gives you a chance to use them by creating Soar agents. Each lesson builds on the previous ones, so it is important to go through them in order. To make the best use of this tutorial, we recommend that you read the tutorial, do the exercises, run the programs, and write your own Soar agents. The programs are available as part of the standard Soar installation. Please use the most recent version. Although the tutorial is long, you should be able to work through it quickly.

What is Soar? We call Soar a unified architecture for developing intelligent systems. That is, Soar provides the fixed computational structures in which knowledge can be encoded and used to produce action in pursuit of goals. In many ways, it is like a programming language, albeit a specialized one. It differs from other programming languages in that it has embedded in it a specific theory of the appropriate primitives underlying reasoning, learning, planning, and other capabilities that we hypothesize are necessary for intelligent behavior. Soar is not an attempt to create a general purpose programming language. You will undoubtedly discover that some computations are difficult or awkward to do in Soar (such as complex math) and they are more appropriately encoded in a programming language such as C, C++, or Java. Our hypothesis is that Soar is appropriate for building autonomous agents that use large bodies of knowledge to generate action in pursuit of goals.

This tutorial is specific to Soar 8, which has significant changes from earlier versions of Soar. These changes improve Soar's ability to maintain the consistency of its reasoning while interacting with dynamic environments.

The tutorial comes in five parts. Part I introduces Soar using a simple puzzle task called the Water Jug to introduce the basic concepts of Soar. This is the classic type of toy problem that people in AI now rail against as being completely unrepresentative of real world problems. Maybe true, but it is simple and easy to understand. After working through Part I, you should be able to write simple Soar programs. Part II uses a Pacman like game called Eaters to introduce interaction with external environments. Part III uses a grid-based tank game called Tank-Soar

and introduces Soar's subgoaling mechanism as it is used for task decomposition. Part IV uses Missionaries and Cannibals problems to further explore internal problem solving and search. Part V uses Missionaries and Cannibals along with the Water Jug to introduce look-ahead planning and learning. Finally, Part VI gives an overview of creating a more complex Soar system for playing Quake.

Soar has its own editor, called Visual Soar, which is highly recommended for use in developing Soar programs. Visual Soar is available from the Soar homepage.

This version of the Tutorial was written with an older version of the TSDebugger, so the displays will not be exactly the same as with the most recent version of Soar.